SAND REPORT

SAND2006-xxx
Unlimited Release
Printed ??? 2005

An Overview of MOOCHO

The Multifunctional Object-Oriented arCHitecture for Optimization

Roscoe A. Bartlett
Optimization/Uncertainty Estim

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-xxx
Unlimited Release
Printed ??? 2005

An Overview of MOOCHO

The Multifunctional Object-Oriented arCHitecture for Optimization

Roscoe A. Bartlett
Department of Optimization and Uncertainty Estimation

Sandia National Laboratories,* Albuquerque NM 87185 USA,

Abstract

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is a C++ Trilinos package of object-oriented software for solving equality and inequality constrained nonlinear programs (NLPs) using large-scale gradient-based optimization methods. The primary focus of MOOCHO up to this point has been the development of active-set and interior-point successive quadratic programming (SQP) methods. MOOCHO was initially developed (under the name rSQP++) to support primarily reduced-space SQP (rSQP) but other related types of optimization algorithms can also be developed. Using MOOCHO, it is possible to specialize all of the linear-algebra computations and also modify many other parts of the algorithm externally (without modifying default library source code). One of the most unique features of the MOOCHO framework is that it supports completely abstract linear algebra which allows sophisticated implementations on parallel distributed-memory supercomputers but is not tied to any particular linear algebra library (although adapters to a few linear algebra libraries are available). In addition, MOOCHO contains adapters to support massively parallel simulation-constrained optimization through Thyra interfaces. Access to a great deal of linear solver technology in Trilinos is available through the "Facade" classes in the Stramikimos package.

This document provides a high-level overview of MOOCHO that describes the motivation for MOOCHO, the basic mathematical notation used in MOOCHO, the algorithms that MOOCHO implements, and what types of optimization problmes are appropriate to be solved by MOOCHO. More detailed documentaion on how to install MOOCHO, how to define NLPs, and how to run MOOCHO algorithms is provided in a companion document [???].

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgment

The authors would like to thank ...

The format of this report is based on information found in [**?**].

Contents

Figures

An Overview of MOOCHO

The Multifunctional Object-Oriented arCHitecture for Optimization

1 Introduction

MOOCHO is an object-oriented C++ software package building gradient-based algorithms for large-scale nonlinear programing. MOOCHO is designed to allow the incorporation of many different algorithms and to allow external configuration of specialized linear-algebra objects such as vectors, matrices and linear solvers (i.e. through Thyra). Data-structure independence has been recognized as an important feature missing in current optimization software [**?**].

While the MOOCHO framework can be used to implement many different types of optimization methods (e.g. Generalized Reduced Gradient (GR) [???], Augmented Lagrangian (AL) [???], Successive Quadratic Programming (SQP) [???] etc.) the main focus has been SQP methods. Successive quadratic programming (SQP) related methods are attractive mainly because they generally require the fewest number of function and gradient evaluations to solve a problem as compared to other optimization methods [**?**]. Another attractive property of SQP methods is that they can be adapted to effectively exploit the structure of the underlying NLP [**?**]. A variation of SQP, known as reduced-space SQP (rSQP), works well for NLPs where there are few degrees of freedom (see Section 2.1) and many constraints. Quasi-Newton methods for approximating the reduced Hessian of the Lagrangian are also very efficient for NLPs with few degrees of freedom. Another advantage of rSQP is that a decomposition for the equality constraints can be used which only requires solves with a basis of the Jacobian of the constraints (see Section 2.3) and therefore can utilize very specialized application-specific data structures and linear solvers. Therefore, rSQP methods can be tailored to exploit the structure of simulation-constrained optimization problems and can show excellent parallel algorithmic scalability.

There is a distiction to be made between a user of MOOCHO and a developer of MOOCHO, though it may it be narrow one in some cases. Here we define a user as anyone who uses MOOCHO to solve an optimization problem using a pre-existing MOOCHO algorithm. A MOOCHO user can vary from someone who uses a predeveloped interface to a modeling environment like AMPL [**?**] to someone who uses MOOCHO to solve a discrietized PDE-constrained optimization problem on a massively parallel computer using specialized application-specific data structures and linear solvers [**?**]. While the first type of user does not need to write any C++ code and does not even need to know what C++ is, the latter type of sophisitcated user has to write a fair amount of C++ code. There are also many different types of use cases of MOOCHO that lie in between these two extremes. This user's guide seeks to address, at least to some degree, the needs of this entire range

of users. Because of this, there will be a fair amount of discussion of the object-oriented design of the relavent parts of MOOCHO.

In the next section (Section 2), the basic mathematical structure of SQP methods is presented. This presentation is intended to establish the nomenclature of MOOCHO for users and developers. This nomenclature is key to being able to understand and modify the MOOCHO algorithms. Appendix **??** contains a summary of this notation. The basic software design of MOOCHO that both users and developers must understand is described in Section **??**. This is followed in Section **??** by a basic description of the linear algebra and NLP interfaces for MOOCHO. These interfaces provide the foundation for allowing the types of specialized data structures and linear solvers that an advanced user would use with MOOCHO. Section **??** discusses a software-based use of MOOCHO for general NLPs where explicit gradient entries are computed. Apart from using a predeveloped interface to MOOCHO (e.g. AMPL), this is the simplest use case for MOOCHO. This section includes a complete example NLP with numerious C++ code excerpts. This discussion is followed up in Section **??** by an example NLP that specializes all of the linear algebra and NLP interfaces, uses application specific linear solvers, and runs on a distributed-memory parallel computer using MPI. This example represents the most advanced use case for MOOCHO and provides the needed foundation for even the most advanced interface to a sophisticated application. Section **??** describes the algorithm configuration classes that are used to build MOOCHO algorithms and includes a fairly detailed discussion of a default configuration called "MamaJama". Details of the input and output files for MOOCHO (for the "MamaJama" configuration and an example NLP) are discussed in Section **??**. This section describes the example printouts that are included in Appendix **??**. Finally, Appendix **??** describes the installation for the base distribution of MOOCHO which is a first step to using MOOCHO.

2 Mathematical Background

2.1 Nonlinear Program (NLP) Formulation

MOOCHO can be used to solve NLPs of the general form:

$$\begin{aligned} \min \quad & f(x) && (1)\\ \text{s.t.} \quad & c(x) = 0 && (2)\\ & x_L \le x \le x_U && (3) \end{aligned}$$

where:

$$x, x_L, x_U \in \mathcal{X}$$

$$
\begin{aligned}
&f(x) : \mathcal{X} \rightarrow \mathbf{R} \\
&c(x) : \mathcal{X} \rightarrow \mathcal{C} \\
&\mathcal{X} \subseteq \mathbf{R}^n \\
&\mathcal{C} \subseteq \mathbf{R}^m.
\end{aligned}
$$

Above, we have been very careful to define vector spaces for the relevant vectors and nonlinear operators. In general, only vectors from the same vector space are compatible and can participate in linear-algebra operations. Mathematically, the only requirement for the compatibility of real-valued vector spaces should be that the dimensions match up and that the same inner products are used [???]. However, having the same dimension and inner product will not always be sufficient to allow the compatibility of vectors from different vector spaces in the implementation (e.g. coeffieients of parallel vectors can have different distributions to processes). Vector spaces become important later when the NLP interfaces and the implementation of MOOCHO is discussed in more detail in Section **??** and in [**?**].

We assume that $f(x)$ and $c_j(x)$ for $j = 1 \ldots m$ in (1)–(2) are nonlinear functions with at least second-order continuous derivatives. The rSQP algorithms described later only require first-order information (derivatives) for $f(x)$ and $c_j(x)$. However, these first derivatives can be provided by finite differences if missing. The simple bound inequality constraints in (3) may have lower bounds equal to $-\infty$ and/or upper bounds equal to $+\infty$. The absences of some of these bounds can be exploited by many of the algorithms.

It is very desirable for the functions $f(x)$ and $c(x)$ to at least be defined (i.e. no `NaN` or `Inf` return values) everywhere in the set defined by the relaxed variable bounds $x_L - \delta \leq x \leq x_U + \delta$. Here, δ (see the method `max_var_bounds_viol()` in the Doxygen documentation for the *`NLP`* interface) is a relaxation (i.e. wiggle room) that the user can set to allow the optimization algorithm to compute $f(x)$ and $c(x)$ outside the strict variable bounds $x_L \leq x \leq x_U$ in order to compute finite differences and the like. The SQP algorithms in MOOCHO will never evaluate $f(x)$ and $c(x)$ outside the above relaxed variable bounds. This gives users a measure of control in how the optimization algorithms interact with the NLP model.

The Lagrangian function $L(\lambda, \nu_L, \nu_U)$ and the Lagrange multipliers (λ, ν_L, ν_U) for this NLP are defined by

$$
\begin{aligned}
L(x, \lambda, \nu_L, \nu_U) &= f(x) + \lambda^T c(x) + \nu_L^T (x_L - x) + \nu_U^T (x - x_U) \ \in \mathbf{R} && (4) \\
\nabla_x L(x, \lambda, \nu) &= \nabla f(x) + \nabla c(x) \lambda + \nu \ \in \mathcal{X} && (5) \\
\nabla^2_{xx} L(x, \lambda) &= \nabla^2 f(x) + \sum_{j=1}^{m} \lambda_{(j)} \nabla^2 c_j(x) \ \in \mathcal{X} | \mathcal{X} && (6)
\end{aligned}
$$

where:

$$
\begin{aligned}
&\nabla f(x) : \mathcal{X} \to \mathcal{X} \\
&\nabla c(x) = \left[\begin{array}{cccc} \nabla c_1(x) & \nabla c_2(x) & \ldots & \nabla c_m(x) \end{array} \right] : \mathcal{X} \to \mathcal{X}|\mathcal{C} \\
&\nabla^2 f(x) : \mathcal{X} \to \mathcal{X}|\mathcal{X} \\
&\nabla^2 c_j(x) : \mathcal{X} \to \mathcal{X}|\mathcal{X} \text{ , for } j = 1 \ldots m \\
&\lambda \in \mathcal{C} \\
&\nu \equiv \nu_U - \nu_L \in \mathcal{X}.
\end{aligned}
$$

Above, we use the notation $\lambda_{(j)}$ with the subscript in parentheses to denote the one-based j^{th} component of the vector λ and to differentiate this from a simple math accent. Also, $\nabla c(x) : \mathcal{X} \to \mathcal{X}|\mathcal{C}$ is used to denote a nonlinear operator (the gradient of the equality constraints $\nabla c(x)$ in this case) that maps from the vector space $\mathcal{X}$ to a linear-operator space $\mathcal{X}|\mathcal{C}$ where the range and the domain are the vector spaces $\mathcal{X}$ and $\mathcal{C}$ respectively. The returned object $A = \nabla c \in \mathcal{X}|\mathcal{C}$ defines a linear operator where $q = Ap$ maps vector from $p \in \mathcal{C}$ to $q \in \mathcal{X}$. The transposed object A^T defines a linear operator where $q = A^T p$ maps vector from $p \in \mathcal{X}$ to $q \in \mathcal{C}$.

Given the definition of the Lagrangian and its derivatives in (4)–(6), the first- and second-order necessary KKT optimality conditions [**?**] for a solution $(x^*, \lambda^*, \nu_L^*, \nu_U^*)$ to (1)–(3) are given in (7)–(13). There are four different categories of optimality conditions: linear dependence of gradients (7), feasibility (8)–(9), non-negativity of Lagrange multipliers for inequalities (10), complementarity (11)–(12), and curvature (13).

$$\nabla_x L(x^*, \lambda^*, \nu^*) = \nabla f(x^*) + \nabla c(x^*)\lambda^* + \nu^* = 0 \tag{7}$$

$$c(x^*) = 0 \tag{8}$$

$$x_L \leq x^* \leq x_U \tag{9}$$

$$(\nu_L)^*, (\nu_U)^* \geq 0 \tag{10}$$

$$(\nu_L)^*_{(i)}((x_L)_{(i)} - (x^*)_{(i)}) = 0, \text{ for } i = 1 \ldots n \tag{11}$$

$$(\nu_U)^*_{(i)}((x^*)_{(i)} - (x_U)_{(i)}) = 0, \text{ for } i = 1 \ldots n \tag{12}$$

$$d^T \nabla^2_{xx} L(x^*, \lambda^*)\, d \geq 0, \text{ for all feasible directions } d \in \mathcal{X}. \tag{13}$$

Sufficient conditions for optimality require that stronger assumptions be made about the NLP (e.g. a constraint qualification on $c(x)$ and perhaps conditions on third-order curvature in case

$$d^T \nabla^2_{xx} L(x^*, \lambda^*)\, d = 0$$

in (13)).

To solve a NLP, an SQP algorithm must first be supplied an initial guess for the unknown variables x_0 and in some cases also initial guesses for the Lagrange multipliers λ_0 and ν_0. The optimization algorithms implemented in MOOCHO generally require that x_0 satisfy the variable bounds in (3), and if not, then the elements of x_0 are forced in bounds.

2.2 Successive Quadratic Programming (SQP)

A popular class of methods for solving NLPs is successive quadratic programming (SQP) [**?**]. An SQP method is equivalent, in many cases, to applying Newton's method to solve the optimality conditions represented by (7)–(8). At each Newton iteration k for (7)–(8), the linear subproblem (also known as the KKT system) takes the form

$$\begin{bmatrix} W & A \\ A^T & \end{bmatrix} \begin{bmatrix} d \\ d_\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_x L \\ c \end{bmatrix} \tag{14}$$

where:

$$\begin{aligned} d &= x_{k+1} - x_k \in \mathcal{X} \\ d_\lambda &= \lambda_{k+1} - \lambda_k \in \mathcal{C} \\ W &= \nabla^2_{xx} L(x_k, \lambda_k) \in \mathcal{X}|\mathcal{X} \\ A &= \nabla c(x_k) \in \mathcal{X}|\mathcal{C} \\ c &= c(x_k) \in \mathcal{C}. \end{aligned}$$

The Newton matrix in (14) is known as the KKT matrix. By substituting $d_\lambda = \lambda_{k+1} - \lambda_k$ into (14) and simplifying, this linear system becomes equivalent to the optimality conditions of the following QP.

$$\begin{aligned} \min \quad & g^T d + \tfrac{1}{2} d^T W d & (15) \\ \text{s.t.} \quad & A^T d + c = 0 & (16) \end{aligned}$$

where:

$$g = \nabla f(x_k) \in \mathcal{X}.$$

The advantage of the QP formulation over the Newton linear system formulation is that inequality constraints can be directly added to the QP and a relaxation can be defined which yields the following QP.

$$\begin{aligned} \min \quad & g^T d + {}^1\!/_2 d^T W d + M(\eta) && (17)\\ \text{s.t.} \quad & A^T d + (1-\eta)c = 0 && (18)\\ & x_L - x_k \le d \le x_U - x_k && (19)\\ & 0 \le \eta \le 1 && (20) \end{aligned}$$

where:

$$M(\eta) \in \mathbf{R} \rightarrow \mathbf{R}.$$

Near the solution of the NLP, the set of optimal active constraints for (17)–(20) will be the same as the optimal active-set for the NLP in (1)–(3) [**?**, Theorem 18.1].

The relaxation of the QP shown in (17)–(20) is only one form of a relaxation but has some essential properties. For example, the solution $\eta = 1$ and $d = 0$ is always feasible by construction. However, the solution $\eta = 1$ and $d = 0$ is of little practical use since it results in zero steps. The penalty function $M(\eta)$ is either linear or quadratic where if $\frac{\partial M(\eta)}{\partial \eta}|_{\eta=0}$ is sufficiently large then an unrelaxed solution (i.e. $\eta = 0$) will be obtained if a feasible region for the original QP exists. For example, the penalty term may take a form such as $M(\eta) = \eta\tilde{M}$ or $M(\eta) = (\eta + {}^1\!/_2\eta^2)\tilde{M}$ where $\tilde{M}$ is a large constant often called "big M". The default QP solver in MOOCHO, QPSchur [???], is careful not to allow the ill-conditioning associated with $\tilde{M}$ to impact the solution unless it is needed for an infeasible QP.

Once a new estimate of the solution $(x_{k+1}, \lambda_{k+1}, \nu_{k+1})$ is computed, the error in the optimality conditions (7)–(9) is checked. If these KKT errors are within some specified tolerance, the algorithm is terminated with the optimal solution. If the KKT error is too large, the NLP functions and gradients are then computed at the new point x_{k+1} and another QP subproblem (17)–(20) is solved which generates another step d and so on. This algorithm is continued until a solution is found or the algorithm runs into trouble (there can be many causes for algorithm failure), or it is prematurely terminated because it is taking too long (i.e. maxumum number of iterations or maximum runtime is exceeded).

The iterates generated from $x_{k+1} = x_k + d$ are generally only guaranteed to converge to a local

solution to the first-order KKT conditions when close to the solution. Therefore, globalization methods are used to insure (given a few, sometimes strong, assumptions are satisfied) the SQP algorithm will converge to a local solution from remote starting points. One popular class of globalization methods are line search methods. In a line search method, once the step d is computed from the QP subproblem, a line search procedure is used to find a step length α such that $x_{k+1} = x_k + \alpha d$ gives *sufficient reduction* in the value of a *merit function* $\phi(x_{k+1}) < \phi(x_k)$. A merit function is used to balance a trade-off between minimizing the objective function $f(x)$ and reducing the error in the constraints $c(x)$. A commonly used merit function is the ℓ_1 (21) where μ is a penalty parameter that is adjusted to insure descent along the SQP step $x_k + \alpha d$ for $\alpha > 0$.

$$\phi_{\ell_1}(x) = f(x) + \mu||c(x)||_1 \tag{21}$$

An alternative line search based on a "Filter" has also been implemented which generally performs better and does not require the maintenance of a penalty parameter μ. Other globalization methods such as trust region (using a merit function or the filter) can also be applied to SQP but no trust region method is currently implemented in MOOCHO.

Because SQP is essentially equivalent to applying Newton's method to the optimality conditions, it can be shown to be quadratically convergent near the solution of the NLP [**?**]. It is this fast rate of convergence that makes SQP the method of choice for many applications. However, there are many theoretical and practical details that need to be considered. One difficulty is that in order to achieve quadratic convergence the exact Hessian of the Lagrangian W is needed, which requires exact second-order information $\nabla^2 f(x)$ and $\nabla^2 c_j(x)$, $j = 1 \ldots m$. For many NLP applications, second derivatives are not readily available and it is too expensive and/or inaccurate to compute them using finite differences. Other difficulties with SQP include how to deal with an indefinite producted Hessian. Also, for large problems, the full QP subproblem in (17)–(20) can be extremely expensive to solve directly. These and other difficulties have motivated the research of large-scale decomposition methods for SQP. One class of these methods is reduced-space (or reduced Hessian) SQP, or rSQP for short.

2.3 Reduced-Space Successive Quadratic Programming (rSQP)

In a reduced-space SQP (rSQP) method, the full-space QP subproblem (17)–(20) is decomposed into two smaller subproblems that, in many cases, are easier to solve. To see how this is done,

first a null-space decomposition [**?**, Section 18.3] is computed for some linearly independent set of the linearized equality constraints $A_d \in \mathcal{X}|\mathcal{C}_d$ where $c_d(x) \in \mathcal{C}_d \in \mathbf{R}^r$ are the decomposed and $c_u(x) \in \mathcal{C}_u \in \mathbf{R}^{(m-r)}$ are the undecomposed equality constraints and

$$c(x) = \begin{bmatrix} c_d(x) \\ c_u(x) \end{bmatrix} \in \mathcal{C}_d \times \mathcal{C}_u \implies \nabla c(x_k) = \begin{bmatrix} \nabla c_d(x_k) & \nabla c_u(x_k) \end{bmatrix} = \begin{bmatrix} A_d & A_u \end{bmatrix} \in \mathcal{X}|(\mathcal{C}_d \times \mathcal{C}_u). \tag{22}$$

Above, the vector space $\mathcal{C} = \mathcal{C}_d \times \mathcal{C}_u$ denotes a blocked vector space (also known as a product space) with a dimension which is the sum of the constituent vector spaces $|\mathcal{C}| = |\mathcal{C}_d| + |\mathcal{C}_u| = r + (m-r) = m$. This decomposition is defined by a null-space linear operator Z and a linear operator Y with the following properties:

$$\begin{array}{ll} Z \in \mathcal{X}|\mathcal{Z} & \text{s.t. } (A_d)^T Z = 0 \\ Y \in \mathcal{X}|\mathcal{Y} & \text{s.t. } \begin{bmatrix} Y & Z \end{bmatrix} \text{ is nonsingular} \end{array} \tag{23}$$

where:

$$\begin{aligned} \mathcal{Z} &\subseteq \mathbf{R}^{(n-r)} \\ \mathcal{Y} &\subseteq \mathbf{R}^r. \end{aligned}$$

It is important to distinguish the vector spaces $\mathcal{Z}$ and $\mathcal{Y}$ from the the linear operators Z and Y. The null-space linear operator $Z \in \mathcal{X}|\mathcal{Z}$ is a linear operator that maps vectors from the space $u \in \mathcal{Z}$ to vectors in the space of the unknowns $v = Zu \in \mathcal{X}$. The linear operator $Y \in \mathcal{X}|\mathcal{Y}$ is a linear operator that maps vectors from the space $u \in \mathcal{Y}$ to vectors in the space of the unknowns $v = Yu \in \mathcal{X}$.

In many presentations of reduced-space SQP, the linear operator Y is referred to as the "range-space" linear operator since several popular choices of this linear operator form a basis for the range space of A_d. However, note that the linear operator Y need not be a true basis linear operator for the range-space of A_d in order to satisfy the nonsingularity property in (23). For this reason, here the linear operator Y will be referred to as the "quasi-range-space" linear operator to make this distinction.

By using (23), the search direction d can be broken down into $d = (1-\eta)Yp_y + Zp_z$, where $p_y \in \mathcal{Y}$ and $p_z \in \mathcal{Z}$ are the known as the quasi-normal (or quasi-range space) and tangential (or null space) steps respectively. By substituting $d = (1-\eta)Yp_y + Zp_z$ into (17)–(20) we obtain the

quasi-normal (24) and tangential (25)–(27) subproblems. In (25), $\zeta \leq 1$ is a damping parameter which can be used to insure descent of the merit function $\phi(x_{k+1} + \alpha d)$.

Quasi-Normal (Quasi-Range-Space) Subproblem

$$p_y = -R^{-1} c_d \in \mathcal{Y} \tag{24}$$

where: $R \equiv [(A_d)^T Y] \in \mathcal{C}_d | \mathcal{Y}$ (nonsingular via (23)).

Tangential (Null-Space) Subproblem (Relaxed)

$$\begin{aligned} \min \quad & (g^r + \zeta w)^T p_z + {}^1\!/_2 p_z^T [Z^T W Z] p_z + M(\eta) && (25) \\ \text{s.t.} \quad & U_z p_z + (1-\eta) u = 0 && (26) \\ & b_L \leq Z p_z - (Y p_y)\eta \leq b_U && (27) \end{aligned}$$

where:

$$\begin{aligned} & g^r \equiv Z^T g \in \mathcal{Z} \\ & w \equiv Z^T W Y p_y \in \mathcal{Z} \\ & \zeta \in \mathbf{R} \\ & U_z \equiv [(A_u)^T Z] \in \mathcal{C}_u | \mathcal{Z} \\ & U_y \equiv [(A_u)^T Y] \in \mathcal{C}_u | \mathcal{Y} \\ & u \equiv U_y p_y + c_u \in \mathcal{C}_u \\ & b_L \equiv x_L - x_k - Y p_y \in \mathcal{X} \\ & b_U \equiv x_U - x_k - Y p_y \in \mathcal{X}. \end{aligned}$$

By using this decomposition, the Lagrange multipliers λ_d for the decomposed equality constraints ($(A_d)^T d + c_d = 0$) do not need to be computed in order to produce steps $d = (1-\eta) Y p_y + Z p_z$. However, these multipliers can be used to determine the penalty parameter μ for the merit function [**?**, page 544] or to compute the Lagrangian function. Alternatively, a multiplier free method for computing μ has been developed and tested with good results [**?**]. In any case, it is useful to compute these multipliers at the solution of the NLP since they give the sensitivity of the objective function

to those constraints [**?**, page 436]. An expression for computing λ_d can be derived by applying (23) to $Y^T \nabla L(x, \lambda, \nu) = 0$ to yield

$$\lambda_d = -R^{-T} \left(Y^T (g + \nu) + U_y^T \lambda_u \right) \in \mathcal{C}_d. \tag{28}$$

There are many details that need to be worked out in order to implement an rSQP algorithm and there are opportunities for a lot of variability. There are some significant decisions that need to be made such as how to compute the null-space decomposition that defines the matrices Z, Y, R, U_z and U_y; and how the reduced Hessian $Z^T W Z$ and the cross term w in (25) are calculated (or approximated).

There are several different ways to compute decomposition matrices Z and Y that satisfy (23) [**?**]. For small-scale rSQP, an orthonormal Z and Y ($Z^T Y = 0$, $Z^T Z = I$, $Y^T Y = I$) can be computed using a QR factorization of A_d [**?**]. This decomposition gives rise to rSQP algorithms with many desirable properties. However, using a QR factorization when A_d is of very large dimension is prohibitively expensive. MOOCHO currently does not implement a orthonormal QR decomposition but one can be added if needed at some point. Other choices for Z and Y have been investigated that are more appropriate for large-scale rSQP. Methods that are more computationally tractable are based on a variable-reduction decomposition [**?**]. In a variable-reduction decomposition, the variables are partitioned into dependent x_D and independent x_I sets

$$x_D \in \mathcal{X}_D \tag{29}$$

$$x_I \in \mathcal{X}_I \tag{30}$$

$$x = \begin{bmatrix} x_D \\ x_I \end{bmatrix} \in \mathcal{X}_D \times \mathcal{X}_I \tag{31}$$

$$\tag{32}$$

where:

$$\mathcal{X}_D \subseteq \mathbf{R}^r$$
$$\mathcal{X}_I \subseteq \mathbf{R}^{n-r}$$

such that the Jacobian of the constraints A^T is partitioned as shown in (33) where C is a square, nonsingular linear operator known as the basis matrix. The variables x_D and x_I are also called the

state and design (or controls) variables [**?**] in some contexts or the basic and nonbasic variables [**?**] in others. What is important about this partitioning of variables is that the x_D variables define the selection of the basis matrix C, nothing more. Some types of optimization algorithms give more significance to this partitioning of variables (for example, in MINOS [**?**] the basic variables are also variables that are not at an active bound) however no extra significance can be attributed here.

This basis selection is used to define a variable-reduction null-space matrix Z in (34) which also determines U_z in (35).

Variable-Reduction Partitioning

$$A^T = \begin{bmatrix} (A_d)^T \\ (A_u)^T \end{bmatrix} = \begin{bmatrix} C & N \\ E & F \end{bmatrix} \tag{33}$$

where:

$C \in \mathcal{C}_d|\mathcal{X}_D$ (nonsingular)
$N \in \mathcal{C}_d|\mathcal{X}_I$
$E \in \mathcal{C}_u|\mathcal{X}_D$
$F \in \mathcal{C}_u|\mathcal{X}_I$.

Variable-Reduction Null-Space Matrix

$$Z \equiv \begin{bmatrix} -C^{-1}N \\ I \end{bmatrix} \tag{34}$$

$$U_z = F - E\,C^{-1}N \tag{35}$$

There are many choices for the quasi-range-space matrix Y that satisfy (23). Two relatively computationally inexpensive choices are the coordinate and orthogonal decompositions shown below.

Coordinate Variable-Reduction Null-Space Decomposition

$$
\begin{aligned}
Y &\equiv \begin{bmatrix} I \\ 0 \end{bmatrix} && (36)\\
R &= C && (37)\\
U_y &= E && (38)
\end{aligned}
$$

Orthogonal Variable-Reduction Null-Space Decomposition

$$
\begin{aligned}
Y &\equiv \begin{bmatrix} I \\ N^T C^{-T} \end{bmatrix} && (39)\\
R &= C(I + C^{-1} N N^T C^{-T}) && (40)\\
U_y &= E - F N^T C^{-T} && (41)
\end{aligned}
$$

The orthogonal decomposition ($Z^T Y = 0$, $Z^T Z \neq I$, $Y^T Y \neq I$) defined in (34)–(35) and (39)–(41) is more numerically stable than the coordinate decomposition defined in (34)–(35) and (36)–(38) and has other desirable properties in the context of rSQP [**?**].

Solutions with linear systems with R in (40) are solved through the formula

$$R^{-1} = (I - D S^{-1} D^T) C^{-1} \tag{42}$$

where $D = -C^{-1}N \in \mathcal{X}_D | \mathcal{X}_I$ and $S = I + D^T D \in \mathcal{X}_I | \mathcal{X}_I$ are explicitly computed, and the symmetric positive definite matrix S is factored using a dense Cholesky method. Therefore, applying R^{-1} only requires a solve with the basis matrix C and applying the factors of S. However, the n_I linear solves needed to form $D = -C^{-1}N$ and the $O((n-r)^2 r)$ dense linear algebra required to compute $D^T D$ can dominate the cost of the algorithm for larger $(n-r)$.

For larger $(n-r)$ if adjoint solves with C^T are available, the coordinate decomposition ($Z^T Y \neq 0$, $Z^T Z \neq I$, $Y^T Y \neq I$) defined in (34)–(35) and (36)–(38) is preferred because it is cheaper but the downside is that it is also more susceptible to problems associated with a poor selection of dependent variables and ill-conditioning in the basis matrix C that can result in greatly degraded performance and even failure of an rSQP algorithm. See the MOOCHO option `quasi_range_space_matrix` in Section **??** for selecting between the orthogonal and the coordinate decompositions.

It is also important to note that MOOCHO can be used to solve nonequality-constrained optimization problems ($m = 0$) and square nonlinear equations ($m = n$). A nonequality-constrained

optimization problem is handled by using $Z = I$ and $Y = \{\text{empty}\}$. A square nonlinear problem is handled using $Z = \{\text{empty}\}$ and $Y = I$. Simpler algorithms are also configured in these two cases.

Another important decision is how to compute the reduced Hessian $Z^T W Z$. For many NLPs, second derivative information is not available to compute the Hessian of the Lagrangian W directly. In these cases, first derivative information can be used to approximate the reduced Hessian $B \approx Z^T W Z$ using quasi-Newton methods (e.g. BFGS) [**?**]. When $(n-r)$ is small, B is small and cheap to update. Under the proper conditions the resulting quasi-Newton, rSQP algorithm has a superlinear rate of local convergence (even using $w = 0$ in (25)) [**?**]. When $(n-r)$ is large, limited-memory quasi-Newton methods can be used, but the price one pays is in only being able to achieve a linear rate of convergence (with a small rate constant hopefully). For some classes of NLPs, good approximations of the Hessian W are available and may have specialized properties (i.e. structure) that makes computing the exact reduced Hessian $B = Z^T W Z$ computationally feasible (i.e. see NMPC in [**?**]). See the options `exact_reduced_hessian` and `quasi_newton` in Section **??**. Other options include solving for system with the exact reduced Hessian $B = Z^T W Z$ iteratively which only requires matrix-vector products with W which can be computed efficiently using automatic differentiation (for instance) in some cases [**?**].

In addition to variations that affect the convergence behavior of the rSQP algorithm, such as null-space decompositions, approximations used for the reduced Hessian and many different types of merit functions and globalization methods, there are also many different implementation options. For example, linear systems such as (24) can be solved using direct or iterative solvers and the reduced QP subproblem in (25)–(27) can be solved using a variety of methods (active set vs. interior point) and software [**?**].

2.4 General Inequalities, Slack Variables and Basis Permutations

Up to this point, only simple variable bounds in (3) have been considered and the SQP and rSQP algorithms have been presented in this context. However, the actual underlying NLP may include general inequalities and take the form

$$\begin{aligned} \min \quad & \breve{f}(\breve{x}) && (43)\\ \text{s.t.} \quad & \breve{c}(\breve{x}) = 0 && (44)\\ & \breve{h}_L \le \breve{h}(\breve{x}) \le \breve{h}_U && (45)\\ & \breve{x}_L \le \breve{x} \le \breve{x}_U && (46) \end{aligned}$$

where:

$$
\begin{array}{l}
\check{x}, \check{x}_L, \check{x}_U \in \check{\mathcal{X}} \\
\check{f}(x) : \check{\mathcal{X}} \to \mathbf{R} \\
\check{c}(x) : \check{\mathcal{X}} \to \check{\mathcal{C}} \\
\check{h}(x) : \check{\mathcal{X}} \to \check{\mathcal{H}} \\
\check{h}_L, \check{h}_L \in \check{\mathcal{H}} \\
\check{\mathcal{X}} \in \mathbf{R}^{\check{n}} \\
\check{\mathcal{C}} \in \mathbf{R}^{\check{m}} \\
\check{\mathcal{H}} \in \mathbf{R}^{\check{m}_I}.
\end{array}
$$

NLPs with general inequalities are converted into the standard form by the addition of slack variables $\check{s}$ (see (50)). After the addition of the slack variables, the concatenated variables and constraints are then permuted (using permutation matrices Q_x and Q_c) according to the current basis selection into the ordering in (1)–(3). The exact mapping from (43)–(46) to (1)–(3) is

$$
\begin{aligned}
x &= Q_x \begin{bmatrix} \check{x} \\ \check{s} \end{bmatrix} && (47) \\
x_L &= Q_x \begin{bmatrix} \check{x}^L \\ \check{h}^L \end{bmatrix} && (48) \\
x_U &= Q_x \begin{bmatrix} \check{x}_u \\ \check{h}_u \end{bmatrix} && (49) \\
c(x) &= Q_c \begin{bmatrix} \check{c}(\check{x}) \\ \check{h}(\check{x}) - \check{s} \end{bmatrix}. && (50)
\end{aligned}
$$

Here we consider the implications of the above transformation in the context of rSQP algorithms.

Note if $Q_x = I$ and $Q_c = I$ that the matrix ∇c takes the form

$$
\nabla c = \begin{bmatrix} \nabla \check{c} & \nabla \check{h} \\ & -I \end{bmatrix} \tag{51}
$$

One question to ask is how the Lagrange multipliers for the original constraints can be extracted from the optimal solution (x, λ, ν) that satisfies the optimality conditions in (7)–(13)? First, consider the linear dependence of gradients optimality condition for the NLP formulation in (43)–(46)

$$
\nabla_{\check{x}} \check{L}(\check{x}^*, \check{\lambda}^*, \check{\lambda}_I^{\check{}\,*}, \check{\nu}^*) = \nabla \check{f}(\check{x}^*) + \nabla \check{c}(\check{x}^*) \check{\lambda}^* + \nabla \check{h}(\check{x}^*) \check{\lambda}_I^{\check{}\,*} + \check{\nu}^* = 0. \tag{52}
$$

To see how the Lagrange multiples λ^* and ν^* can be used to compute $\check{\lambda}^*$, $\check{\lambda}_I^{\ *}$ and $\check{\nu}^*$ one simply has to substitute (47) and (50) with $Q_x = I$ and $Q_c = I$, for instance, into (7) and expand as follows

$$\begin{aligned}
\nabla_x L(x,\lambda,\nu) &= \nabla f + \nabla c\lambda + \nu \\
&= \begin{bmatrix} \nabla \check{f} \\ 0 \end{bmatrix} + \begin{bmatrix} \nabla \check{c} & \nabla \check{h} \\ & -I \end{bmatrix} \begin{bmatrix} \lambda_{\check{c}} \\ \lambda_{\check{h}} \end{bmatrix} + \begin{bmatrix} \nu_{\check{x}} \\ \nu_{\check{s}} \end{bmatrix} \\
&= \begin{bmatrix} \nabla \check{f} + \nabla \check{c}\lambda_{\check{c}} + \nabla \check{h}\lambda_{\check{h}} + \nu_{\check{x}} \\ -\lambda_{\check{h}} + \nu_{\check{s}} \end{bmatrix}. \qquad (53)
\end{aligned}$$

By comparing (52) and (53) it is clear that the mapping is $\check{\lambda} = \lambda_{\check{c}}$, $\check{\lambda}_I = \lambda_{\check{h}} = \nu_{\check{s}}$ and $\check{\nu} = \nu_{\check{x}}$. For arbitrary Q_x and Q_c it is also easy to perform the mapping of the solution. What is interesting about (53) is that it says that for general inequalities $\check{h}_j(\check{x})$ that are not active at the solution (i.e. $(\nu_{\check{s}})_{(j)} = 0$), the Lagrange multiplier for the converted equality constraint $(\lambda_{\check{h}})_{(j)}$ will be zero. This means that these converted inequalities can be eliminated from the problem and not impact the solution (which is what one would have expected). Zero multiplier values means that constraints will not impact the optimality conditions or the Hessian of the Lagrangian.

The basis selection shown in (22) and (31) is determined by the permutation matrices Q_x and Q_c and these permutation matrices can be partitioned as

$$Q_x = \begin{bmatrix} Q_{xD} \\ Q_{xI} \end{bmatrix} \qquad (54)$$

$$Q_c = \begin{bmatrix} Q_{cD} \\ Q_{cU} \end{bmatrix}. \qquad (55)$$

A valid basis selection can always be determined by simply including all of the slacks $\check{s}$ in the full basis and then finding a sub-basis for $\nabla \check{c}$. To show how this can be done, suppose that $\nabla \check{c}$ is full column rank and the permutation matrix $(\check{Q}^x)^T = \begin{bmatrix} (\check{Q}_{xD})^T & (\check{Q}_{xI})^T \end{bmatrix}$ selects a basis $\check{C} = (\nabla \check{c})^T (\check{Q}_{xD})^T$. Then the basis selection for the transformed NLP (with $Q_c = I$)

$$Q_x = \begin{bmatrix} \check{Q}_{xD} & & \\ & & I \\ & \check{Q}_{xI} & \end{bmatrix} \qquad (56)$$

$$C = \begin{bmatrix} (\check{Q}_{xD}\nabla \check{c})^T & \\ (\check{Q}_{xD}\nabla \check{h})^T & -I \end{bmatrix} \qquad (57)$$

$$N = \begin{bmatrix} (\check{Q}_{xI}\nabla \check{c})^T \\ (\check{Q}_{xI}\nabla \check{h})^T \end{bmatrix} \qquad (58)$$

could always be used regardless of the properties or implementation of $\nabla \breve{h}$.

Notice that basis matrix in (57) is lower block triangular with non-singular blocks on the diagonal. It is therefore straightforward to solve for linear systems with this basis matrix. In fact, the direct sensitivity matrix $D = C^{-1}N$ takes the form

$$D = -\begin{bmatrix} (\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^{T} \\ (\breve{Q}_{xD}\nabla\breve{h})^{T}(\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^{T} - (\breve{Q}_{xI}\nabla\breve{h})^{T} \end{bmatrix}. \tag{59}$$

Note that if the forward sensitivities $(\breve{Q}_{xD}\nabla\breve{c})^{-T}(\breve{Q}_{xI}\nabla\breve{c})^{T}$ are computed up front then this is little extra cost in forming this decomposition. The structure of (59) is significant in the context of active-set QP solvers that solve the reduced QP subproblem in (25)–(27) using a variable-reduction null-space decomposition. When an implicit adjoint method is used, a row of D corresponding to a general inequality constraint only has to be computed if the slack for the constraint is at a bound. Also note that the above transformation does not increase the total number of degrees of freedom of the NLP since $n - m = \breve{n} - \breve{m}$. All of this means that adding general inequalities to a NLP imparts little extra cost for an active-set rSQP algorithm if the forward/direct sensitivity method is used or if these constraints are not active when using the adjoint method.

For reasons of stability and algorithm efficiency, it may be desirable to keep at least some of the slack variables out of the basis and this can be accommodated also but is more complex to describe.

Most of the steps in an SQP algorithm do not need to know that there are general inequalities in the underlying NLP formulation but some steps do (i.e. globalization methods and basis selection). Therefore, those steps in an SQP algorithm that need access to this information are allowed more detailed access of the underlying NLP in a limited manner.

3 Overview of Software Architecture, Solvers, and Examples

4 Defining Optimization Problems

4.1 Defining general serial NLPs with explicit derivative entries

4.2 Defining simulation-constrained parallel NLPs through Thyra

5 Solving Optimization Problems with MOOCHO

5.1 Algorithm configurations for MOOCHO

5.2 Running MOOCHO algorithms

5.3 Summary